ES6、ES7、Webpack和Babel

第1天课堂笔记

讲师：邵山欢

日期：2017年11月14日

# ES6、ES7简介和调试方法

## 简介

ECMAScript6 就是ECMAScript2015

ECMAScript7 就是ECMAScript2016

ECMAScript8 就是ECMAScript2017

ECMA组织2015年开始每年都要发布一个ECMAScript的新版本。

ECMAScript现在的语言实现就两个：JavaScript、ActionScript。

## 1.2 用浏览器调试

使用最新版本的浏览器，就能调试一些ES6、7、8的语法。



在浏览器调试的时候，要有HTML的壳子，要有script标签：

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Document</title>

</head>

<body>

<script type="text/javascript">

var [a,b,c] = [2,3,4];

console.log(a);

console.log(b);

console.log(c);

</script>

</body>

</html>

兼容性可以访问ECMAScript官网，会有一个兼容性汇总。

## 1.3 使用NodeJS调试



现在8.7.0是开发者预览版版本中比较新的版本。

写一些ES6语法的程序：

var [a,b,c] = [1,2,3];

console.log(a);

console.log(b);

console.log(c);

var obj1 = {

a : 1,

b : 2,

c : 3

}

var obj2 = {

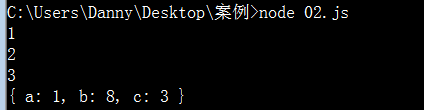
...obj1 ,

b : 8

}

console.log(obj2);

node跑：



## 1.4 使用Babel翻译

babel是我们学习的第二个nodejs工作流工具，需要用-g安装（第一个是cnpm）。

npm -g install babel-cli

-g安装的安装的东西，我们叫做CLI程序，commond line interface，命令行接口。这些命令可以在CMD窗口中使用。

因为我们npm -g install cnpm --registry http://npm.taobao.org/

所以我们能够在CMD中运行cnpm

因为我们npm -g babel-cli

所以我们能够在CMD中运行babel



首先-g安装babel

npm install -g babel-cli

然后检查babel是否安装成功：



现在，在你的项目的根目录创建一个文件，叫做.babelrc文件。

rc表示resource资源的意思。

.babelrc文件windows下不允许文件名以.开头：



所以要曲线救国，创建一个随便名字的文件：

然后rename命令：

rename haha.txt .babelrc

此时.babelrc生成了。

这个文件这么写：

{

"presets" : ["es2015"]

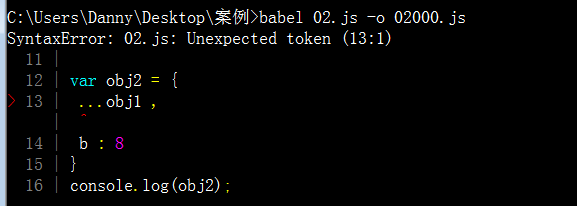
}

设置完.babelrc之后，还要安装这个preset的依赖：

cnpm intall --save-dev babel-preset-es2015

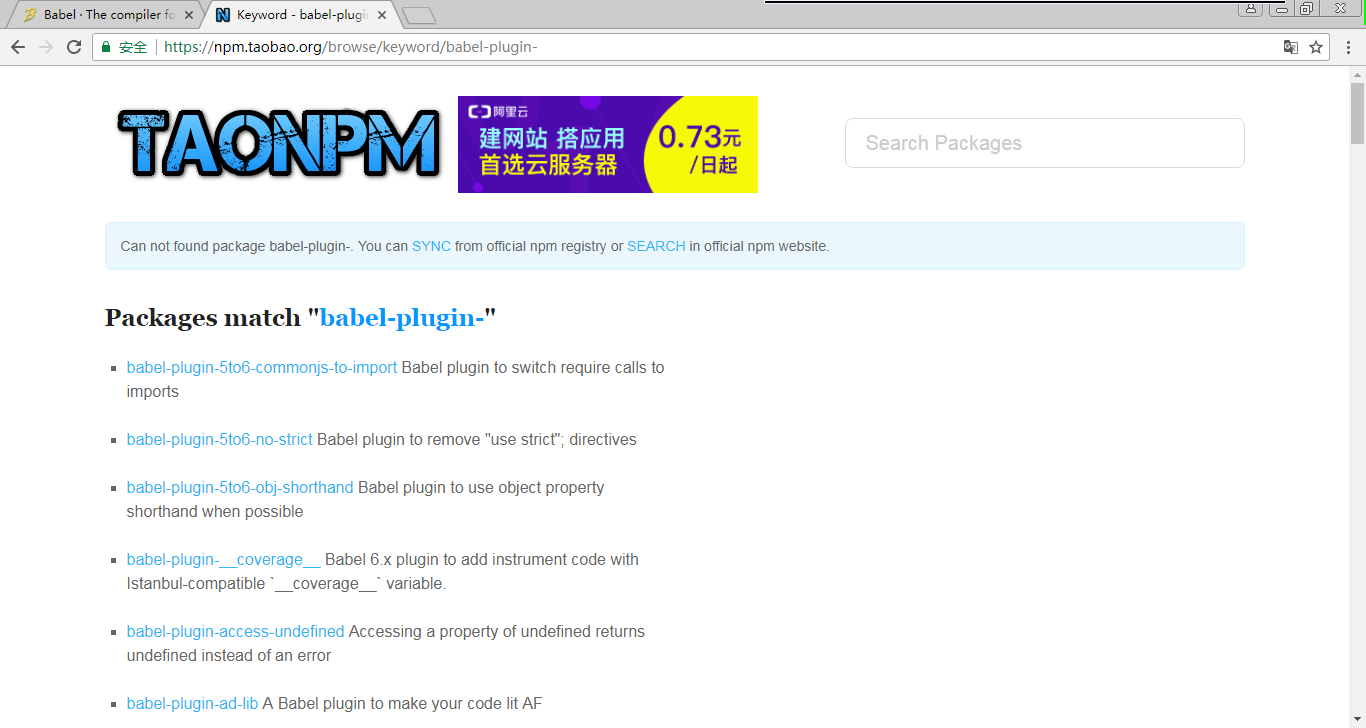
--save-dev表示在项目的package.json文件中的“devDependencies”字段列出，表示项目的开发依赖。就是说是在项目开发的时候，用的翻译器、打包工具、构建工具等等。

babel不认识...语法，会报错



所以要给babel安装插件，让babel能够识别更多的语法。

babel的插件在npm库中都叫做babel-plugin-\*\*\*：



安装插件：

cnpm install --save-dev babel-plugin-transform-object-rest-spread

更改.babelrc文件，告诉babel我们要求你使用插件：

{

"presets" : ["es2015"],

"plugins" : ["transform-object-rest-spread"]

}

随后就可以用balbel命令进行翻译了：

babel 02.js -o 02000.js

表示将02.js文件翻译为02000.js。

# const和let

## 2.1 const

手册是阮一峰的博客，网址见下图：

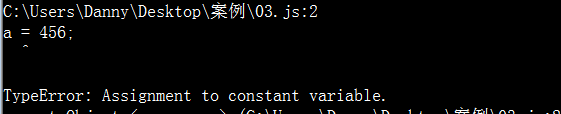


const声明一个只读的常量。一旦声明，常量的值就不能改变。通常用const定义函数。

下面的程序是错误的，因为a是常量，不能更改：

const a = 123;

a = 456;



什么东西不变呢？比如圆周率：

**const PI = 3.14159226;**

console.log(PI);

常量一般来说所有字母要大写。

**函数的定义一般来说都是不变的，所以今后的函数一律用const定义**。

**const sum** = function(a,b){

return a + b;

}

console.log(sum(2,3));

## 2.2 let

let用来定义**块级作用域**变量。它的用法类似于var，**但是所声明的变量，只在let命令所在的代码块内有效**。

所谓的块级作用域指的是大括号、所有的语句体。

**{**

**var** a = 1;

**}**

console.log(a); //能够输出1

**{**

**let** b = 2;

**}**

console.log(b); //报错

var定义的变量出了{}对儿依然有定义，但是let没有定义了。

今后，我们的所有的循环变量将用let来定义：

for(let i = 0 ; i < 10 ; i++){

}

console.log(i); //报错

需要注意：

1） 不管是let还是const，都没有了变量声明的提升；

console.log(m); //报错，没有变量声明的提升；

let m = 100;

1. 不管是let还是const，babel一律翻译为var。

# 变量的自动解构和剩余参数

## 3.1 自动解构

只有数组和对象能够自动解构。

ES6中允许从数组中提取值，按照对应位置，对变量赋值。对象也是一样的。

|  |  |
| --- | --- |
| **数组的结构** | **对象的解构** |
| var **[**a,b,c**]** = [1,2,3];  console.log(a);  console.log(b);  console.log(c); | var **{**a,b,c**}** = {"a":1,"b":2,"c":3};  console.log(a);  console.log(b);  console.log(c); |

**对象的结构往往用在函数的形参列表中**，调用函数的时候，传的对象（实参）就能够自动解构。

function People(**{**name,age,sex,yuwen,shuxue,yingyu**}**){

this.name = name;

this.age = age;

this.sex = sex;

this.yuwen = yuwen;

this.shuxue = shuxue;

this.yingyu = yingyu;

}

var xiaoming = new People(**{**

**"name" : "小明",**

**"age" : 12,**

**"sex" : "男",**

**"yuwen" : 34,**

**"shuxue" : 44,**

**"yingyu" : 66**

**}**);

console.log(xiaoming);

## 3.2 剩余参数

左边迎接的人群，比来宾少，有些来宾就接不到：

var [a,b,c] = [1,2,3,4,5,6,7];

console.log(a); //1

console.log(b); //2

console.log(c); //3



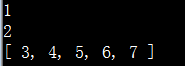
我们可以在c变量之前加上...运算符，表示c现在接受所有剩余的参数。

var [a,b,**...c**] = [1,2,3,4,5,6,7];

console.log(a);

console.log(b);

console.log(c); //[3,4,5,6,7]



**需要注意的事情是：...只能出现在最后一项。**

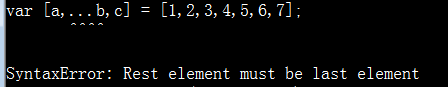
错误的：

~~var [a,...b,c] = [1,2,3,4,5,6,7];~~

console.log(a);

console.log(b); //报错！

console.log(c);



## 3.3 强制解构

在数组前面加上...运算符，表示强制解构。**这个数组将变为零散量**。

var arr = [1,2,3];

console.log(**...arr**);

console.log(1,2,3);

如同每个量之间隔了空格输出了。



**...运算符也可以作用于对象，通常用于一个情况：创建obj1的副本，仅仅改变obj1的某一（些）个属性**。

var obj1 = {

"a" : 1,

"b" : 2,

"c" : 3

};

var obj2 = {

**...obj1** ,

"b" : 8

};

console.log(obj2);



强制解构还可以用于类数组对象，可以将类数组对象变为真的数组：

const fun = function(){

**//下面的语句是一个奇淫技巧，是最快的将类数组对象变为数组的方法**

var arr = **[...arguments]**;

console.log(arr); //[ 1, 2, 3, 4, 5, 6, 7, 8 ]

console.log(Array.isArray(arr)); //true

}

fun(1,2,3,4,5,6,7,8);



如果用babel解构对象，需要安装babel的插件babel-plugin-transform-object-rest-spread。

小题目：

var [[a,b],[c,d],...e] = [[1,2],[3,4,5],6,7,8,9];

console.log(a); //1

console.log(b); //2

console.log(c); //3

console.log(d); //4

console.log(e); //[6,7,8,9]

# 数组的新方法

## 4.1 forEach()

长久以来我们用for(var i = 0 ; i < arr.length ; i ++){}来遍历数组。

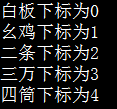
但是今天，我们有了forEach()函数，今天开始再也不用for循环遍历数组了！

var arr = ["白板","幺鸡","二条","三万","四筒"];

**arr.forEach(**function(item , index){

console.log(item + "下标为" + index);

}**)**;



注意参数是个函数（蓝色底），这个函数中有两个参数，分别是item和index，是遍历到的这一项的内容和下标。

## 4.2 Map()

**Map()返回一个新的数组，新数组的每一项是原数组的映射。**

比如我们的要求是创建一个数组，每一项都是原来的数组的两倍。

var arr = [43,4,23,23,34,23];

var arr2 = **arr.map(**function(item){

return item \* 2;

}**)**;

console.log(arr2);



注意的是：map()出的数组，一定和原数组长度相同！

函数里面的的return就是新数组中这一项的数值。

## 4.3 filter()

**从一个数组中，过滤一些项组合成为新数组，此时使用filter()函数。**

比如我们想把arr数组中所有的偶数都提炼出来组合成为新的数组：

var arr = [3,43,25,43,543,645,765,7,23,432,4,324,23,43,45];

var arr2 = **arr.filter(function(item){**

**return item % 2 == 0;**

**})**;

console.log(arr2);

filter()中有一个函数，这个函数return为true的项会被加入到新的数组中。

再比如从一个数组中，提取所有及格组合成为新数组：

var arr = [

{"name":"小明" , "fenshu" :66},

{"name":"小红" , "fenshu" :16},

{"name":"小强" , "fenshu" :26},

{"name":"小刚" , "fenshu" :86}

];

var arr2 = arr.filter(function(item){

return item.fenshu >= 60;

});

console.log(arr2);



## 4.4 reduce()

reduce表示迭代遍历，每一项的遍历都可以使用遍历上一项的时候的结果。

reduce可以理解为“降阶”。

reduce里面的函数有两个参数，分别是a、b。

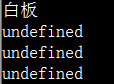
我们来探究a、b什么意思。所以我们直接输出a：

var arr = ["白板","幺鸡","二条","三万","四筒"];

arr.reduce(function(a,b){

console.log(a);

});



你会发现少了一项。

var arr = ["白板","幺鸡","二条","三万","四筒"];

arr.reduce(function(a,b){

console.log(b);

});



reduce的机理：

* a就表示遍历前一项的时候return的值，b表示这一项的值。
* 系统会自动从下标为1的项开始遍历，遍历下标为1的项的时候，a的值是第0项。
* 最后的一项的return，就是总return。

例子1：求数组的和

var arr = [3,4,5,2,2];

var result = arr.reduce(function(a,b){

return a + b;

});

console.log(result); //16

例子2：求数组的最大值

var arr = [4,43,5,436,45,623,4,324,23,443,53,65,46,45,6,234,324];

var result = **arr.reduce(function(a,b){**

**return a >= b ? a : b;**

**})**;

console.log(result); //623

reduce的哲学意义，就是把一个数组揉吧揉吧，每一项都参与参与，得出唯一的一个数字。

## 4.5 （重要）综合运用map、filter、reduce、结构

现在给大家一个难题，

var xiaoming = {

"name" : "小明",

"age" : 12,

"sex" : "男",

"friends" : [

{

"name" : "小红",

"age" : 13

},

{

"name" : "小强",

"age" : 14

},

{

"name" : "小刚炮",

"age" : 18

}

]

}

**● 题目1：不允许更改小明！返回一个新对象，这个对象和小明相同，仅让小明的年龄变为15岁！**

答案：

var xiaoming = {

"name" : "小明",

"age" : 12,

"sex" : "男",

"friends" : [

{

"name" : "小红",

"age" : 13

},

{

"name" : "小强",

"age" : 14

},

{

"name" : "小刚炮",

"age" : 18

}

]

}

var xiaoming2 = {

**...xiaoming ,**

"age" : 15

}

**● 题目2：不允许更改小明！让小强的age变为22。**

var xiaoming2 = {

...xiaoming ,

"friends" : xiaoming.friends.map(function(item){

//如果这一项的name是小强，此时返回别的项都不改，只改变age项

if(item.name == "小强"){

return {

...item ,

"age" : 22

}

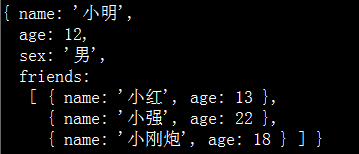
}

//反之，返回原来的项

return item;

});

}



**● 题目3：不允许更改小明！让小明的只有15岁以下的朋友。**

var xiaoming2 = {

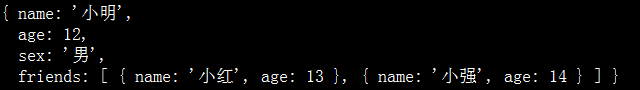
...xiaoming ,

"friends" : xiaoming.friends.filter(function(item){

return item.age < 15;

})

}



**● 题目4：不允许更改小明！小明增加一个朋友，老王，88岁**

var xiaoming2 = {

...xiaoming ,

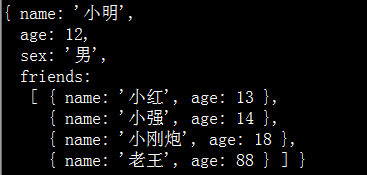
"friends" : [...xiaoming.friends , {

"name" : "老王" ,

"age" : 88

}]

}



var xiaoming2 = {

...xiaoming ,

"friends" : [{

"name" : "老王" ,

"age" : 88

},...xiaoming.friends ]

}

**● 题目5：不允许更改小明！增加一个朋友并且自动编号：**

var xiaoming2 = {

...xiaoming ,

"friends" : [

...xiaoming.friends,

{

"id" : xiaoming.friends.reduce(function(a,b){

return a.id > b.id ? a : b;

}).id + 1 ,

"name" : "老王" ,

"age" : 88

}

]

}

不更改原来的对象，返回一个新对象，新对象是原来对象的增、删、改某些属性，这叫做函数式编程，也叫作兰姆达式编程。

中午的题目讲解：

原来的数据：

var canting = {

"name" : "全聚德",

"cai" : {

"liangcai" : [

{

"name" : "凉拌西红柿",

"price" : 15

},

{

"name" : "拍黄瓜",

"price" : 18

}

],

"recai" : [

{

"name" : "宫保鸡丁",

"price" : 25

},

{

"name" : "红烧肉",

"price" : 45

},

{

"name" : "辣子鸡",

"price" : 35

}

]

}

}

删除用filter

改变用map

增加用...

操作：

// ① 宫保鸡丁不卖了

var canting1 = {

...canting ,

"cai" : {

...canting.cai ,

"recai" : canting.cai.recai.filter(function(item){

return item.name != "宫保鸡丁";

})

}

}

// ② 宫保鸡丁售价变为26

var canting2 = {

...canting ,

"cai" : {

...canting.cai ,

"recai" : canting.cai.recai.map(function(item){

if(item.name == "宫保鸡丁") {

return {

...item ,

"price" : 26

}

}

return item;

})

}

}

// ③ 增加一个新菜：红烧茄子，19元

var canting3 = {

...canting ,

"cai" : {

...canting.cai ,

"recai" : [

...canting.cai.recai ,

{

"name" : "红烧茄子" ,

"price" : 19

}

]

}

}

// ④ 售价20以上的热菜都不卖了

var canting4 = {

...canting ,

"cai" : {

...canting.cai ,

"recai" : canting.cai.recai.filter(function(item){

return item.price <= 20

})

}

}

学习这个很关键，比如react的组件中有state（状态）：

this.state = {

a : 1,

b : [5,6,7,8],

c : 3

};

修改的时候，不能更改原来的state。比如我们要把9加入到b属性中，正确：

this.setState({

b : [...this.state.b , 9]

});

## 4.6 includes()方法

Array.prototype.includes方法返回一个布尔值，表示某个数组是否包含给定的值。之前我们判断数组中是否存在一项，此时用indexOf() != -1（IE8开始兼容）。如果要IE6、7兼容，必须一项一项遍历比较。

ES6中，有了更简单的比较方法：

var arr = ["三星","oppe","苹果"];

console.log(arr.includes("vivo")); //false

console.log(arr.includes("苹果")); //true



注意，includes是全等比较，带着类型比较的：

var arr = ["三星","oppe","苹果",7];

console.log(arr.includes("7")); //false

## 4.7 Array.from()方法

Array.from方法用于将两类对象转为真正的数组。

var obj = {

"0" : "甲",

"1" : "乙",

"2" : "丙",

"length" : 3

};

**var realObj3 = Array.from(obj);**

console.log(realObj3);

# 对象的改变

## 5.1 省略v

当一个对象的k、v相同（k的名字和v的变量名相同），此时可以省略v。

比如下面有三个变量a、b、c，要作为一个对象的属性名a、b、c，同时值也是1、2、3。因为k、v一致，此时就可以省略v：

var a = 1;

var b = 2;

var c = 3;

var obj = {

a ,

b ,

c

}

console.log(obj);

这种省略v的写法非常常见：

var result = (function(){

var s = 0;

for (var i = 0; i <= 100; i++) {

s += i;

};

return s;

})();

var json = {result};

console.log(json);



注意，省略v的时候，k不能有引号！

var obj = {

a

}

不是：

var = {

"a"

}

## 5.2 对象的方法现在可以简写

var obj = {

a : 1,

fn : function(){

console.log(this.a);

}

}

obj.fn();

现在简写为：

var obj = {

a : 1,

fn(){

console.log(this.a);

}

}

obj.fn();

## 5.3 Object.keys()和Object.values()

**Object.keys()返回对象的键名数组；**

**Object.values()返回对象的值数组；**

var obj = {

"k0" : "甲",

"k1" : "乙",

"k2" : "丙"

};

console.log(Object.keys(obj));

console.log(Object.values(obj));



## 5.4 Object.is()

ES5 比较两个值是否相等，只有两个运算符：相等运算符（==）和严格相等运算符（===）。它们都有缺点，前者会自动转换数据类型，后者的NaN不等于自身，以及+0等于-0。**JavaScript 缺乏一种运算，在所有环境中，只要两个值是一样的，它们就应该相等。**

**console.log(Object.is(NaN , NaN)); //true**

console.log(Object.is(+0 , -0)); //false

比较的时候比较的仍然是对象是否是同一个对象。

var arr1 = [1,2,3];

var arr2 = [1,2,3];

console.log(Object.is(arr1 , arr2)); //false

因为arr1和arr2在内存中是两个不同的数组。

## 5.5 Object.assign

**Object.assign方法用于对象的合并**，将源对象（source）的所有可枚举属性，复制到目标对象（target）。

var obj1 = {"a" :1 , "b" : 2 , "c" : 3};

var obj2 = {"b" :4 , "c" : 5 , "d" : 6};

var obj3 = {"c" :7 , "d" : 8 , "e" : 9};

**Object.assign(obj1 , obj2 , obj2);**

console.log(obj1);



注意Object.assign()可以有无限个参数，但是只会更改第一个参数对象。

## 5.6 对象中键加方括号 - 动态键

当对象的键名是一个变量的时候，在{}的里面，可以写[]包裹这个变量成为键名。

var sex ="男";

if(sex == "男"){

var k = "feihuoliang";

}else{

var k = "xiongwei";

}

var obj = {

"name" : "小明",

"sex" : sex,

**[k] : 130**

}

console.log(obj);



# 函数的改变

## 6.1 箭头函数

下面的sum就是标准的箭头函数，省略掉了function这个关键字：

const sum = (a,b) => {

return a + b;

}

console.log(sum(3,4));

=>是一个完整的运算符，之间不能有空格。

**如果函数体内只有一行语句，可以省略{}和return**。

**const sum = (a,b) => a + b;**

console.log(sum(45,5));

**反过来，如果函数体内有多条语句，此时必须写{}和return单词。**

**箭头函数的美，最能体现在连续箭头**。当外层函数被调用的时候，将返回内层的函数。调用的时候可以加两层圆括号。

const fn = **(a , b) => (m) =>** {

return a + b + m;

}

console.log(fn(3,4)(5)); //12

等价于：

const fn = **function(a,b)**{

return **function(m)**{

return a + b + c

}

}

**箭头函数中的this是定义时所处的上下文，和如何调用无关！**

比如下面有一个obj对象，这个对象的b方法是用箭头函数写的。这个箭头函数所在的最外层函数体是window域。所以this不管b函数如何被调用，一律是window对象。

var a = 9;

var obj = {

a : 1 ,

b : () => {

console.log(this.a);

}

}

obj.b(); //9 ， 因为虽然是对象打点调用函数，但是由于函数是箭头函数，所以之前学习的规则2,不生效。一律上下文是window对象。

甚至call、apply失效：

var haha = {

a : 8888

}

obj.b.call(haha); //9

obj.b.apply(haha); //9

this在箭头函数中是固定的，这个性质是我们应该防止的、加以警惕的；而不是利用的。

箭头函数有几个使用注意点。

（1）函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。

（2）不可以当作构造函数，也就是说，不可以使用new命令，否则会抛出一个错误。

（3）不可以使用arguments对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。

1. 不可以使用yield命令，因此箭头函数不能用作 Generator 函数。

## 6.2 bind()

bind表示给一个函数定死上下文，而无论它如何被调用。

比如下面有一个函数fun，函数体输出this.a。

我们在后面有一个.bind()语句，彻底将fun函数的上下文定死了，就是obj对象。

无论fun函数如何被调用，一律上下文是obj对象。

function fun(){

console.log(this.a);

}

//这是一个对象

var obj = {

a : 233 ,

b : 886

}

//这是另一个对象

var another = {

a : 6666666666666

}

//定死上下文为obj对象

fun = fun.bind(obj);

fun(); //233

fun.call(another); //233，call失效了

fun.apply(another); //233，apply失效了

bind和call和apply有什么区别？

* bind不调用函数，call和apply会调用函数。
* bind能绑死上下文，call和apply是临时的。

## 6.3 默认参数

在形参列表中可以加等号运算符，表示参数的默认参数，当我们没有传入这个参数的时候，将自动使用这个默认参数。

const mianji = (r , **pi = 3.14**) => {

return pi \* r \* r;

}

console.log(mianji(10 , Math.PI)); //传了第二个参数，就是用第二个参数

console.log(mianji(10)); //没有第二个参数，将用默认的3.14

# 字符串、正则改变

建议大家可以自己学习：

http://es6.ruanyifeng.com/#docs/string

http://es6.ruanyifeng.com/#docs/regex

比如：





``是新增的定界符，可以和${变量}使用，减少连字符的使用：

`我是一个${this.name}，我今年${this.age}岁了`

# 类的改变

现在可以用class关键字来定义一个类，语法：

class People{

//构造函数

constructor(name , age , sex){

this.name = name;

this.age = age;

this.sex = sex;

}

changge(){

console.log("我是一个" + this.name + "今年" + this.age + "岁啦！！");

}

goDie(){

console.log("死啦");

}

}

var xiaoming = new People("小明",12,"男");

console.log(xiaoming);



语法提示：



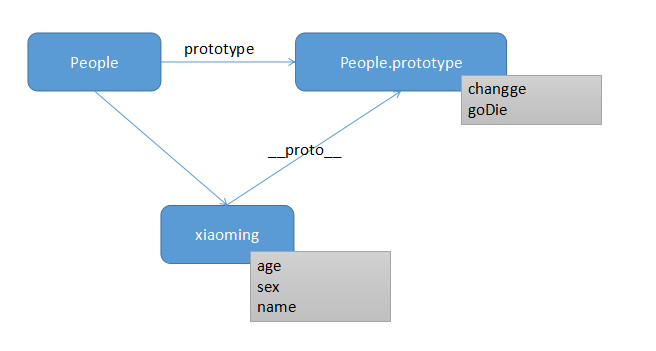
虽然增加了class关键字，但是JS中还是没有类！JS中仍然是简单的“基于对象”原型链的模式来模拟类的。

也就是说，机理没有变化！

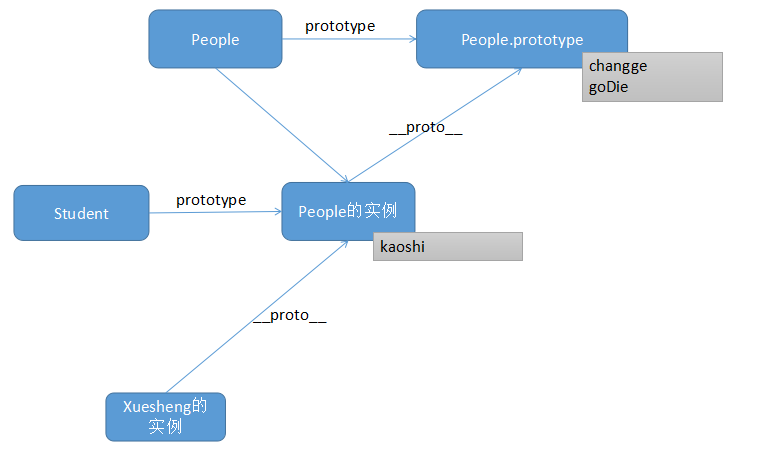
console.log(xiaoming.changge === People.prototype.changge); //true

console.log(xiaoming.hasOwnProperty("changge")); //false

console.log(xiaoming.hasOwnProperty("name")); //true



继承的实现：



复习一下原来继承的实现：

function People(name,age,sex){

this.name = name;

this.age = age;

this.sex = sex;

}

People.prototype.changge = function(){

console.log(`我是一个${this.name}，我今年${this.age}岁了`);

}

function Student(name,age,sex,xuehao,banji){

this.name = name;

this.age = age;

this.sex = sex;

this.xuehao = xuehao;

this.banji = banji;

}

Student.prototype = new People();

Student.prototype.kaoshi = function(){

console.log(`${this.name}在考试`);

}

var xiaohua = new Student("小花",12,"女",10001,"1班");

xiaohua.changge();

xiaohua.kaoshi();

ES6中简化了类的继承：

class People{

constructor(name , age , sex){

this.name = name;

this.age = age;

this.sex = sex;

}

changge(){

console.log("我是一个" + this.name + "今年" + this.age + "岁啦！！");

}

goDie(){

console.log("死啦");

}

}

class Student **extends** People{

constructor(name , age , sex , xuehao , banji){

super(name , age , sex); //调用超类的构造器

this.xuehao = xuehao;

this.banji = banji;

}

kaoshi(){

console.log(`${this.name}在考试`);

}

}

var xiaohua = new Student("小花",12,"女",10001,"1班");

xiaohua.changge();

xiaohua.kaoshi();

# Promise对象

回调套回调是很恶心的。我们有三个文件1.txt 、2.txt、3.txt。要求用fs.readFile()方法顺序读取三个文件，此时：

var fs = require("fs");

fs.readFile("./txt/1.txt",**(err,data)=>{**

console.log(data.toString());

fs.readFile("./txt/2.txt",**(err,data)=>{**

console.log(data.toString());

fs.readFile("./txt/3.txt",**(err,data)=>{**

console.log(data.toString());

**}**);

**}**);

**}**);

回调套回调，因为readFile是异步函数。

现在我们要学习的知识，就是将“回调套回调”这种回调黑洞（callback blackhole）变得优雅。

## 9.1 ES6中的最基本的Promise案例

Promise表示“承诺”。

* **Promise在使用的时候，一定是作为一个函数的return值返回的**。下面的duwenjian（读文件）函数就return了一个Promise对象。这个函数称之为“Promise函数”
* **resolve 表示成功的时候调用的函数，resolve是谁呢？是你then的时候逗号前的函数。**
* **reject表示失败的时候调用的函数，reject是谁呢？是你then的时候逗号后的函数。**
* **在new Promise((resolve,reject)=>{})函数里面要写具体的异步义务。**
* **then表示“然后”，在then()、then()、then()中往往是return一个duwenjian函数。**

var fs = require("fs");

function duwenjian(url){

return new Promise((resolve , reject) => {

fs.readFile(url , (err , data) => {

if(err){

reject(err);

return;

}

resolve(data);

});

});

}

duwenjian("./txt/1231231.txt")

**.then(**

(data)=>{

console.log(data.toString());

**return duwenjian("./txt/2.txt");**

}**,**

(err)=>{

console.log("读取失败");

}

**)**

**.then**((data)=>{

console.log(data.toString());

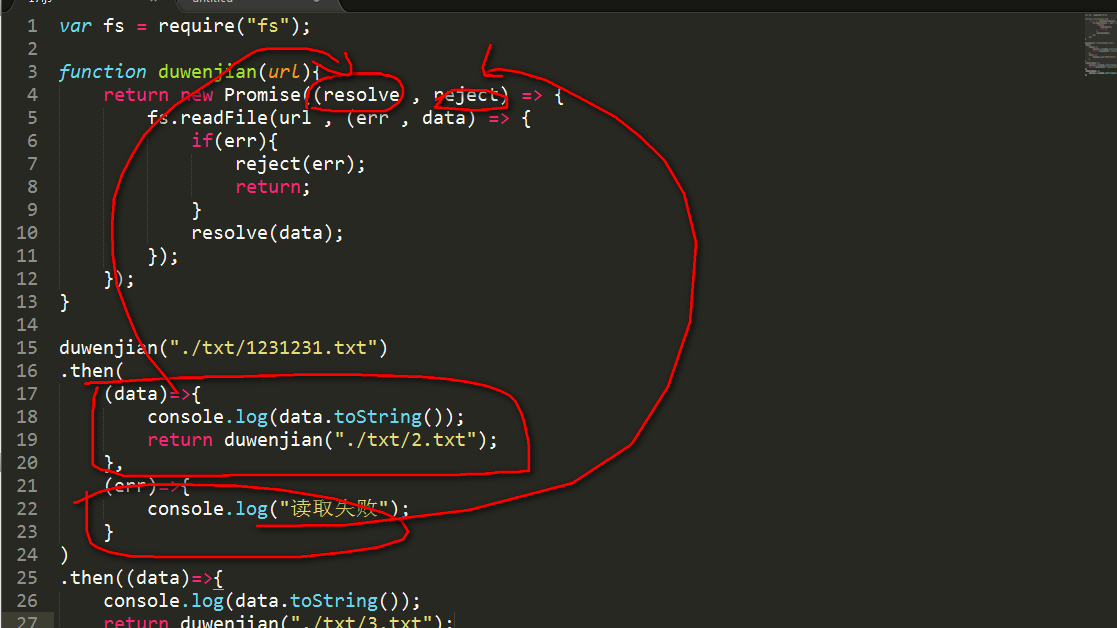
**return duwenjian("./txt/3.txt");**

}**)**

**.then(**(data)=>{

console.log(data.toString());

}**)**



Promise的总结：原来的回调套回调，现在通过：

1. 给函数加一个return new Promise((resolve,reject)=>{})壳
2. 调用的时候增加.then()

轻松把回调套回调变为了火车。

Promise本质上是JavaScript中新的语法糖

语法糖：指计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。通常来说使用语法糖能够增加程序的可读性，从而减少程序代码出错的机会。

## 9.2 ES7中的async和await

ES6中确实优雅的解决了回调黑洞的问题，但是尼玛火车黑洞来了。

ES7变得更优雅。

**首先，那个返回Promise对象的函数，没有任何的变化！下面duwenjian函数没有发生任何变化**！

我们做一个主函数，比如叫做main()，**给这个函数叫上async关键字**，表示这个函数里面有异步的语句需要等待。异步语句的前面，加上await字样，**表示这条语句是异步语句**，请你等我做完之后再做后面的语句。

var fs = require("fs");

function duwenjian(url){

return new Promise((resolve , reject) => {

fs.readFile(url , (err , data) => {

resolve(data.toString());

});

});

}

//异步函数

**async** function main(){

var data1 = **await** duwenjian("./txt/1.txt");

console.log(data1);

var data2 = **await** duwenjian("./txt/2.txt");

console.log(data2);

var data3 = **await** duwenjian("./txt/3.txt");

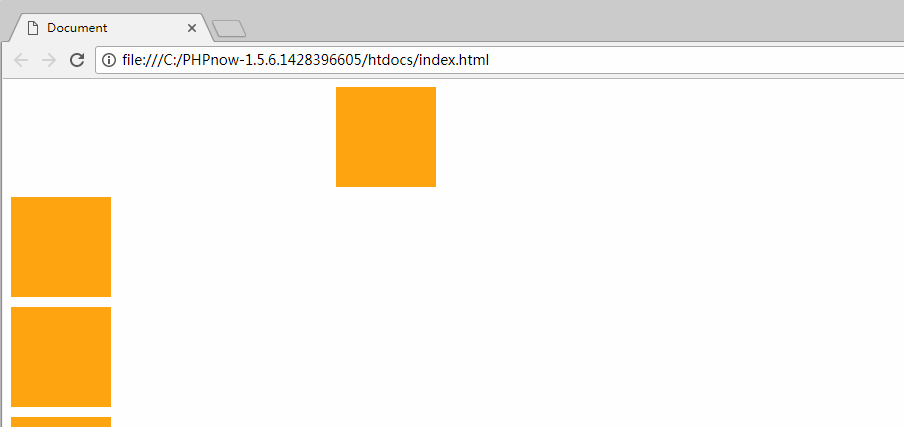
console.log(data3);

}

main();

**async、await最牛的在于：将异步语句变为了同步语句！**

我们用jQuery中的animate()函数来做例子，页面上4个盒子，要求依次动画，一个动完一个动。



传统：

$("#box1").animate({"left":1000},1000,function(){

$("#box2").animate({"left":1000},1000,function(){

$("#box3").animate({"left":1000},1000,function(){

$("#box4").animate({"left":1000},1000,function(){

});

});

});

});

现在我们可以用async和await结合Promise函数来轻松解决回调黑洞：

<script type="text/javascript">

function dong(id){

**return new Promise((resolve)=>{**

$(id).animate({"left":1000},1000,resolve);

**});**

}

async function main(){

await dong("#box1");

await dong("#box2");

await dong("#box3");

await dong("#box4");

}

main();

</script>

需要注意的是：

1. 要用async、await，必须事先有一个Promise产生器；
2. 写await语句的函数之前必须加async标记；
3. await后面必须跟着一个Promise函数的调用;
4. 在Promise对象的(resolve,reject)=>{}函数里面，必须有resolve，表示此异步函数已经执行完毕。
5. await等号左边将自动成为resolve函数中的参数。

<script type="text/javascript">

function duwenjian(url){

return new Promise((resolve , reject) => {

$.get(url , function(data){

resolve(JSON.parse(data));

});

});

}

async function main(){

var data1 = await duwenjian("txt/1.txt").then(data=>data.result);

var data2 = await duwenjian("txt/2.txt").then(data=>data.result);

var data3 = await duwenjian("txt/3.txt").then(data=>data.result);

console.log(data1);

console.log(data2);

console.log(data3);

}

main();

</script>

最后说一点，就是awaite的函数也可以有then()，then()里return的东西，会被等号左边接收。

<script type="text/javascript">

function duwenjian(url){

**return new Promise((resolve , reject) => {**

$.get(url , function(data){

resolve(JSON.parse(data));

});

**});**

}

**async** function main(){

var data1 = **await** duwenjian("txt/1.txt").then(data=>data.result);

var data2 = **await** duwenjian("txt/2.txt").then(data=>data.result);

var data3 = **await** duwenjian("txt/3.txt").then(data=>data.result);

console.log(data1);

console.log(data2);

console.log(data3);

}

main();

</script>

## 9.3 fetch()

**fetch是ES8中定稿的一个功能，是一个Ajax函数的升级版。天生可以跨域，不需要jQuery。**

**fetch()函数的最大优点是返回一个Promise对象**，所以就可以结合then()、async、await轻松解决回调套回调的问题。

<script type="text/javascript">

async function main(){

var data1 = await **fetch(**"./txt/1.txt").then(data=>data.json()**)**;

var data2 = await **fetch(**"./txt/2.txt").then(data=>data.json()**)**;

var data3 = await **fetch(**"./txt/3.txt").then(data=>data.json()**)**;

console.log(data1);

console.log(data2);

console.log(data3);

}

main();

</script>

fetch函数也可以发POST请求：

fetch("/search/project/", {

method: "POST",

headers: {

'Content-Type': 'application/json'

},

body: "JSON的字符串形式"

}).then(function(response) {

});

今天学习的所有特性的兼容性，可以在https://www.caniuse.com/查询。